# Path Selection Based Acceleration of Conditionals in CGRAs

ShriHari RajendranRadhika, Aviral Shrivastava, Mahdi Hamzeh
Arizona State University, Tempe, AZ, USA 85281
Email: {shrihari, aviral.shrivastava, mahdi}@asu.edu

***Abstract*—Coarse Grain Reconfigurable Arrays (CGRAs) are promising accelerators capable of achieving high performance at low power consumption. While CGRAs can efficiently accelerate loop kernels, accelerating loops with control flow (loops with if-then-else structures) is quite challenging. Existing techniques use predication to handle control flow execution – in which they execute operations from both the paths, but commit only the result of operations from the path taken by branch at run time. However, this results in inefficient resource usage and therefore poor mapping and lower acceleration. The state-of-the-art dual issue scheme fetches instructions from both the paths, but executes only the ones from the correct path but this scheme has an overhead in instruction fetch bandwidth. In this paper, we propose a solution in which after resolving the branching condition, we fetch and execute instructions only from the path taken by branch. Experimental results show that our solution achieves 34.6% better performance and 52.1% lower energy consumption on an average compared to state of the art dual issue scheme.**

## I. Introduction

Accelerators are now widely accepted as an inseparable part of computing fabric. Special purpose, custom hardware accelerators have been shown to achieve the highest performance with the least power consumption [1]. However, they are not programmable and incur a high design cost. On the other hand Graphics Processing Units or GPUs, although programmable, are limited to accelerating only *parallel loops* [2]. Field Programmable Gate Arrays (FPGAs) have some of the advantages of hardware accelerators and are also programmable [3]. However, their *fine-grain* reconfigurability incurs a very high cost in terms of energy efficiency [4].

Coarse Grain Reconfigurable Arrays (CGRAs) are programmable accelerators that promise high performance at low power consumption [5] [6]. CGRA is an array of processing elements (PE) which are connected with each other through an interconnection network as shown in Figure 1. Each PE consist of a functional unit, local register files and output register. The functional unit typically performs arithmetic, logic, shift and comparison operations. The operands for each PE can be obtained from neighbouring PEs, its own output from previous cycle, data bus or the local register file. Every cycle, instructions are issued to all PEs specifying the operation and the position of input operands. CGRAs are more power-efficient than FPGAs, since they are programmable at a coarser granularity – at the level of arithmetic operations – in contrast to FPGAs which are programmable at bit level. Since CGRAs support both parallel and pipelined execution, they can accelerate both parallel and non-parallel loops [5] – as opposed to GPUs that can only accelerate parallel loops.

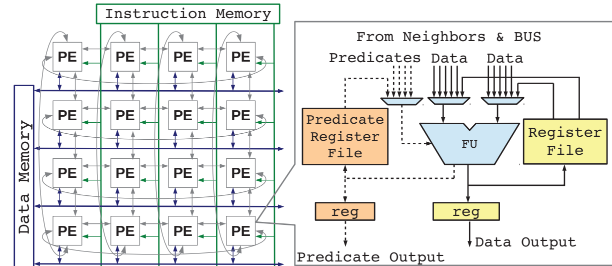



Fig. 1. A $4 \times 4$ CGRA with PEs connected in a torus interconnect. A PE consists of an ALU and register files and receives an instruction each cycle to operate on available data.

One of the major challenges associated with CGRAs is that of accelerating loops with **if-then-else** structures. Hamzeh et al.[7] show the importance of accelerating loops with if-the-else constructs because they are present in many long running loops in important applications. Since the result of the conditional is known only at run time, existing solutions in CGRAs handle them by predication [8], [9], [10], [11] where instructions are executed from both the paths of an if-then-else structure and then commit the results of only the instructions from the path taken by the branch at run time. While predication allows for correct execution, it results in inefficient resource usage – and therefore inefficient execution. Dual-issue schemes [12], [13], [7] try to improve this by fetching the instructions from both paths but only executing instructions from the correct path. They achieve higher performance, but at the cost of increased instruction fetch bandwidth – they have to fetch 2 instructions per PE every cycle.

In this paper, we propose to accelerate loops with if-then-elses by fetching and executing instructions only from the path taken by branch at run time. Our solution contains two parts, i) execute the branch condition as early as possible, and ii) once the branch is computed, communicate its results to the Instruction Fetch Unit (IFU) of the CGRA, which then starts to fetch instructions from the correct path. Experimental results on accelerating loop kernels, with if-then-else structures from biobench [14] and SPEC [15] benchmark by our solution results in 34.6% improvement in performance and 52.1% lower energy consumption (CGRA power and power spent on instruction fetch operation) as compared to state of the art dual-issue technique presented in [7].

978-3-9815370-4-8/DATE15/©2015 EDAA

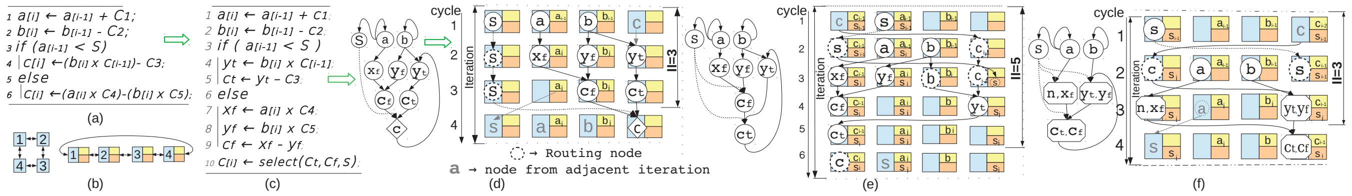



Fig. 2. (a) Shows a loop kernel with control flow (b) Shows a flattened 2x2 CGRA with data and predicate output registers highlighted (c) Shows the loop in (a) after SSA transformation, $C$ represents constants available from immediate field of an instruction to PE, (d)(e) and (f) Shows the Data Flow Graph (DFG) of the loop kernel mapped on the CGRA via partial predication scheme, full predication and dual issue scheme respectively.

## II. Background and Related Work

Loop kernels are the most desirable parts of the program to be accelerated in a CGRA [16]. Most of the computational loop kernels have if-then-else structures in them[7]. Consider a loop kernel with if-then-else as shown in figure 2(a),(c). The kernel has 5 predicate based instructions, two in the **if-block** and three in the **else-block**. The variable $c[i]$ is updated in both the blocks, so it must be conditionally updated depending upon the branch outcome at runtime. Variables $y_t$ and $x_f$ , $y_f$ are intermediate variables used for the computation of $c_t$ and $c_f$ in $if$ and $else$ block respectively. Three commonly used techniques to execute loop kernels with if-else structures are: i) Partial predication, ii) Full predication, and iii) Dual issue.

In a partial predication scheme [13][8], the if-path and else-path operations of a conditional branch are executed in parallel in different PE resources. The final result is selected between outputs of two paths based on outcome of the conditional operation (predicate value) as shown in figure 2(d). This is accomplished by a select instruction (shown as a diamond shaped node for variable $c[i]$ in Fig.2(d)) which acts like a hardware multiplexer or a $phi$ operation in compilers. PE template for a partial predication scheme is shown in figure 1. There is a predicate mux selecting a predicate available from the neighbouring PEs or from the predicate register file or the predicate value generated by the PE in previous cycle. Predicate communication is done via a predicate register and a predication network. Fig.2(d) shows how the loop kernel in Fig.2(a) can be executed via a partial predication scheme. The metric of performance is the Initiation Interval (II), which is the number of cycles after which the next iteration can be started and hence lower the better. Obtained II is 3 in this case.

In full predication scheme [10],[13], the output of false path operations are suppressed based on a predicate bit (0 for false path operations). Operations that update the same variable have to be mapped to the same PE albeit at different cycles. Figure 2(e) shows that operations $c_t$ and $c_f$ are mapped to the same PE (PE 1) at cycles 4 and 5 which has the predicate value. The correct value is available in the register of the PE (PE 1) after the execution of operations from both paths is past (in our case, at cycle 6). This eliminates the need for select instructions. Hardware support requires a predicate enabled PE output. Achieved II = 5 for our example fig.2(e).

In dual-issue[12], each PE receives two instructions, one from the if-path and the other from else-path at each cycle. At run-time, the PE executes only one of the instructions based on the predicate bit. Since an operation from the false path is not executed, a select operation is not required. Operations in the different paths producing the same output (e.g., $c_t$ and $c_f$) are merged together to execute on the same PE. Nodes that have 2 instructions associated with them are called merged nodes, as shown by octagons in fig.2(f). In addition to the architectural support required for partial predication scheme, supporting Dual issue requires a 2x1 mux which selects either the $if$-path operation or the $else$-path operation to be executed by the PE. Achieved II = 3 as shown in fig.2(f).

## III. Inefficiencies of existing techniques

The fundamental inefficiency of existing solutions in handling loops with control flow is that they do not utilize the knowledge of the branch outcome to reduce the overhead of branch execution – even after the branch outcome is known. For instance, the branch outcome is known at cycle 1 in the partial and full predication schemes (figs. 2(d), and 2(e)). However, they still execute three unnecessary operations, $x_f$, $y_f$ and $c_f$, if the condition evaluates to true. This blind-eye towards an important output and failure to use its result translates into excessive resource usage, lower performance and more dynamic power wasted. This limitation may be tolerable for if-then-else structures which have relatively lower number of operations, but it becomes high for if-then-elses where the number of operations in the conditional path is quite large. Even though the dual-issue scheme does not execute the $false$ path instructions it still keeps on fetching them – not utilising the branch outcome even after it is known in cycle 1 (fig. 2(f)).

The other limitation of the existing approaches is that the predicate value must be communicated to the PEs executing the $if$-path and the $else$-path operation. This communication is done either by storing the predicate value in the internal register of a PE or through the predicate network via routing. The need for this communication results in restrictions on where the conditional operations can be mapped. For instance, in partial predication, the select operation $c$ can be mapped only to PEs in which the corresponding predicate value is available, and in full predication scheme, the operations $c_t$, and $c_f$ should be mapped onto the same PE (PE1) where the predicate value is available. For dual issue scheme, the predicate value must be present in the internal register of the PEs executing merged nodes $\langle nop, x_f \rangle$, $\langle y_t, y_f \rangle$ and $\langle c_t, c_f \rangle$ to select the right instruction. These restrictions in mapping conditional operations lead to poor resource utilization.

|  | PE 1 | PE 2 | PE 3 | PE 4 |
|---|---|---|---|---|
| 1 | <idle> | <$blt\ a_{[i-1]}, S \mid 2$ > | <idle> | <idle> |
| 2 | <idle> | <$a_{[i]} \leftarrow a_{[i-1]} + C_1$> | <$b_{[i]} \leftarrow b_{[i-1]} - C_2$> | <idle> |
| 3 | F: <$x_f \leftarrow a_{[i]} + C_4$> | <idle> | <idle> | <$y_f \leftarrow b_{[i]} \times C_5$> |
| 4 | <idle> | <idle> | <idle> | <$c_f \leftarrow x_f - y_f$ > |
| 5 | T: <nop> | <idle> | <idle> | <$y_t \leftarrow b_{[i]} \times c_{[i-1]}$> |
| 6 | <idle> | <idle> | <idle> | <$c_t \leftarrow y_t - C_3$> |

Fig. 3. CGRA instructions for execution of the kernel via PSB technique

# IV. Our Approach

Considering that only one path is taken at run time for the if-then-else construct, we communicate the predicate (result of the branch instruction) to the Instruction Fetch Unit (IFU) of the CGRA, to selectively issue instructions only from the path taken by the branch at runtime. This is the essence of our Path Selection based Branch (PSB) technique. This is similar to if-then-else execution in general purpose processors, but while simultaneously taking advantage of parallelism available in the CGRA for performance improvement through software pipelining.

Figure 3 shows the arrangement of instructions of the loop body in figure 2(c) to be executed on a 2x2 CGRA as per our approach. In the first cycle, the branch operation $\langle blt\ a[i-1],\ S \mid 2\rangle$ is executed on PE 2, while the rest of the PEs are idle. The operation $\langle blt\ a, b \mid K\rangle$ is a branch instruction that compares if $a < b$. $K$ is the maximum number of cycles required to execute the if-path or the else-path. The $else$-path is composed of instructions at addresses 3 and 4, and it takes 2 cycles to execute. The $if$-path also takes 2 cycles, and is composed of instructions at addresses 5 and 6. Even though the condition in the branch operation executes in cycle 1, the operations in the $if$-path or $else$-path does not begin execution until cycle 3. Cycle 2 is the delay slot of the CGRA, in which the operations independent of the current branch outcome (including operations from adjacent iterations) can be executed. This delay slot cycle is used to communicate the branch outcome to the IFU. Operations $\langle a[i] = a[i-1]+C1\rangle$ and $\langle b[i] = b[i-1]-C2\rangle$ are executed on PEs 2 and 3 in the delay slot in cycle 2. After the delay slot the IFU will start issuing instructions from the path taken by the branch. If the $else$-path is taken, then instructions 3 and 4 will be issued. After executing $else$-path instructions, the IFU will skip the next $K$ instructions, and start issuing instructions after that. If the branch is taken, then the IFU will skip $K$ instructions and start issuing $if$-path instructions.

For branch outcome based issuing of instructions, additional hardware support is required as shown in figure 4. The architecture of partial predication scheme is extended to communicate the branch outcome to CGRA′s IFU along with the information of number of cycles to execute the branch. IFU is modified to issue instructions from the path taken based on branch information (outcome + no.of cycles for conditional path).

## A. What must the compiler do?

In addition to the hardware support, the compiler must map operations from the loop kernel (including if-path, else-path and select or phi operations) onto the PEs of the time-extended CGRA. The PEs required to map the if-then-else portion of the loop kernel is the union of the PEs on which the operations

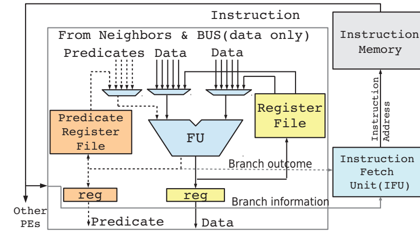


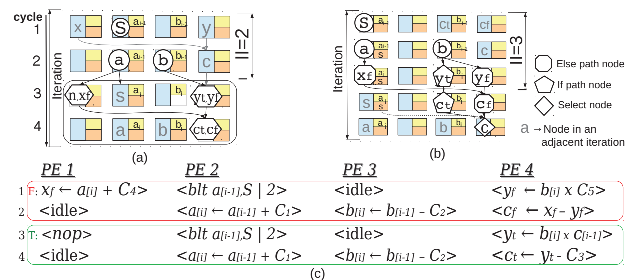

Fig. 4. Architectural support for our proposed approach. The branch information and outcome is communicated to the instruction fetch unit (IFU) to issue instructions only from the path taken at run time.

from the if and else-path are mapped. Since only one of the paths is taken at runtime, we map the operations from the if-path and the operations from the else-path to the same PEs, so that the number of PEs used to map the if-then-else is equal to the maximum of the number of PEs required to map either path's operations as shown in fig. 5(a). Hence, irrespective of the path taken by branch, the PEs that are allocated paired operations from the if and else-path, executes a useful operation from the path taken. This results in better utilization of PE resources and more PEs being available to map operations from adjacent iterations to facilitate the use of a modulo scheduling scheme to further improve the performance. Let us consider a case where if-path and else-path operations are mapped onto different PEs, fig.5(b), the PEs mapped with if-path operations will be inactive when else-path executes and vice-versa. In such a case, the PEs allocated to execute the operations in the conditional path is the sum of the PEs required for the if-path operations and else-path operations. But at run time only the PEs which were mapped with operations from the path taken is active and PEs associated with the false path is inactive, resulting in higher II (II = 3) fig.5(b).

Hence, by pairing operations from if-path and from else-path to form a fused node and mapping them to a CGRA via a modulo scheduling scheme, we achieve lower II = 2, which is the best achieved so far, and therefore better performance and resource utilization. The mapping and the corresponding instruction arrangement is shown in fig. 5(a) and fig. 5(c).

## B. Problem Formulation

Since pairing of operations from the if-path and the else-path improves resource utilization and performance, we define our problem formulation as obtaining a valid pairing ensuring the correct functionality of the loop kernel. Problem is




|  | PE 1 | PE 2 | PE 3 | PE 4 |
|---|---|---|---|---|
| 1 | F: $x_f \leftarrow a_{[i]} + C_4$> | <$blt\ a_{[i-1]}, S \mid 2$> | <idle> | <$y_f \leftarrow b_{[i]} \times C_5$> |
| 2 | <idle> | <$a_{[i]} \leftarrow a_{[i-1]} + C_1$> | <$b_{[i]} \leftarrow b_{[i-1]} - C_2$> | <$c_f \leftarrow x_f - y_f$> |
| 3 | T: <nop> | <$blt\ a_{[i-1]}, S \mid 2$> | <idle> | <$y_t \leftarrow b_{[i]} \times c_{[i-1]}$> |
| 4 | <idle> | <$a_{[i]} \leftarrow a_{[i-1]} + C_1$> | <$b_{[i]} \leftarrow b_{[i-1]} - C_2$> | <$c_t \leftarrow y_t - C_3$> |

(c)

Fig. 5. (a) Shows a mapping with pairing of operations via PSB resulting in lower II (b) Shows mapping without pairing resulting in poor resource utilization (higher II), (c) Instructions after pairing and modulo scheduling.

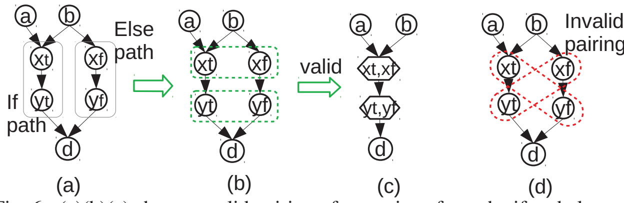


Fig. 6. (a)(b)(c) shows a valid pairing of operations from the if and else-path. (d) shows an invalid pairing since such a pairing fails to meet the criteria for validity and a feasible schedule for such a pairing does not exist.

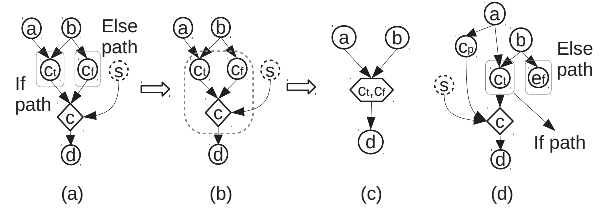


Fig. 7. (a)(b)(c) Shows elimination of eligible phi/select operation with inputs from if-path and else-path, (d) shows an example of a phi that cannot be eliminated since its input does not belong to the set of if or else-path operations.

formulated as finding a transformation $T(D) = P$ from the input Data Flow Graph (DFG): D = $(N, E)$ to an output DFG: $P(M, R)$ with fused nodes, with the objective of minimizing $|M|$ (N and M represent the set of nodes in D and $P$) while retaining the correct functionality.

**Input:** DFG: $D = (N, E)$ is a data flow graph that represents the loop kernel, where the set of vertices $N$ are the operations in the loop kernel, and for any two vertices, $u, v \in N, e = (u, v) \in E$ iff the operation corresponding to $v$ is data dependent or predicate dependent on the operation $u$. For a loop with control flow $N = \{N_{if} \cup N_{else} \cup N_{other}\}$ where $\{N_{if}\}$ is the set of nodes representing the operations in the if-path and likewise $\{N_{else}\}$ for the else-path. $N_{other}$ is the set of nodes representing operations not in the $if$ or the else-path and includes select operations.

**Output:** DFG: $P = (M, R)$: Where $M$ is the set of nodes in the transformed DFG representing the operations in the loop kernel with $M = \{M_{fused} \cup M_{other}\}$. The nodes $M_{fused}$ represent the fused nodes. Each fused node $m \in M_{fused}$ is a tuple $m = \langle m_{if}, m_{else} \rangle$, where $m_{if} \in N_{if} \cup \{nop\}$ and $m_{else} \in N_{else} \cup \{nop\}$. For nodes $x, y \in M_{fused}, r = (x, y) \in R$ iff there is an edge $e_{if} = (x_{if}, y_{if}) \in E$ or an edge $e_{else} = (x_{else}, y_{else}) \in E$. For nodes $x_{other} \in M_{other}, y \in M_{fused}, r = (x_{other}, y) \in R$ iff there is an edge $e_{if} = (x_{other}, y_{if}) \in E$ or an edge $e_{else} = (x_{other}, y_{else}) \in E$ where $x_{other} \in N_{other}$. For nodes $x \in M_{fused}, y_{other} \in M_{other}, r = (x, y_{other}) \in R$ iff there is an edge $e_{if} = (x_{if}, y_{other}) \in E$ or an edge $e_{else} = (x_{else}, y_{other}) \in E$ where $y_{other} \in N_{other}$.

**Valid Output:** The output DFG $P$ obtained after transformation is valid iff: For two vertices x,y with $x = (x_{if}, x_{else}), y = (y_{if}, y_{else}) \in M_{fused}$ and $r = (x, y) \in R$ then if there is a path from $x_{if}$ *to* $y_{if}$ then there is no path (intra-iteration) from $y_{else}$ *to* $x_{else}$ and if there is a path from $x_{else}$ *to* $y_{else}$ there is no path (intra-iteration) from $y_{if}$ *to* $x_{if}$ originally in the input DFG. However, recurrence paths satisfying inter iteration dependencies are valid. Figure 6 shows an example each for a valid paring (6(b),(c)) and an invalid pairing (6(d)).

**Optimization:** Objective is to minimize $|M|$ under constraints of a valid output. $|M_{fused}|$ can be minimised by minimizing no.of nops used to make a pair. $|M_{other}|$ can be minimised by eliminating the eligible select or phi operations that belong to $N_{other}$.

*Select/Phi operation elimination:* A select operation is used to select an output of a variable updated in both paths. If the if-path operation and the else-path operation updating the same variable is paired to form a fused node, there is no need for a select operation since at run time only one of the operations is executed, the output of the fused node has the right value after execution. Figure 7 shows scenarios in which a select/phi operation can be eliminated.

### C. Our Heuristic

The process of creating a DFG from CFG (Control Flow Graph) of a loop is presented in [17]. The operations from the if-path and else-path form the set of operations $N_{if}$ and $N_{else}$ respectively. The algorithm for forming the DFG with fused node is shown in Alg.1. Fig. 8 demonstrates how the kernel in fig. 2(c) is transformed using PSB. The algorithm starts with pairing of operations from if and else-path. Pairing starts from the terminating operations $c_t$ and $c_f$ in the if-path and the else-path respectively, lines 1,2 in alg. 1. Then the pairing proceeds iteratively in a partial order of operations as long as there are unpaired operations in the if and the else-path. This partial order is according to the dependence flow of the operations in the if block and the else block of the CFG. Node $\langle y_t, y_f \rangle$ represents a resulting fused node after iterative pairing. If the operations in the if and else-path are unbalanced, the unbalanced operations are paired with a $nop$, lines 7,9 in Alg. 1, hence, the unpaired else-path operation $x_f$ is paired

**Algorithm 1**: PSB (Input $DFG(D)$, Output $DFG(P)$)

```
1  n_if ← getLastNode({N_if});
2  n_else ← getLastNode({N_else});
3  while (n_if ≠ NULL or n_else ≠ NULL) do
4      if n_if ∈ N_if and n_else ∈ N_else then
5          fuse(n_if, n_else);
6      else if n_if ∈ N_if and n_else == NULL then
7          fuse(n_if, nop);
8      else if n_if == NULL and n_else ∈ N_else then
9          fuse(nop, n_else);
10     n_if ← getLastRemainingNode({N_if});
11     n_else ← getLastRemainingNode({N_else});
12 for n_i such that i=0 to |N| do
13     if n_i is an eligible select operation ∈ N_other, ∋
       input_1(n_i), input_2(n_i) = m_fused ∈ M_fused then
14         Eliminate_phi(n_i);
15 Remove_Redundant_Arcs(E);
16 Prune_Predicate_Arcs(E);
```

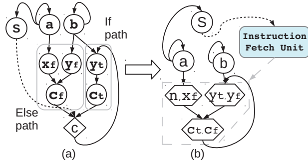


Fig. 8. Shows construction of DFG with fused nodes from an input DFG.

with a nop to form a fused node $\langle nop, x_f \rangle$. After all operations in the if and else path are paired, eligible select operations are eliminated via a phi elimination pass, line 14 in Alg. 1. In this example, the phi operation $c$ is eligible for eliminated and the output of the fused node $\langle c_t, c_f \rangle$ serves as the output of the eliminated phi node. Then the redundant edges are eliminated and predicate arcs are pruned and final output DFG (P) is obtained as shown in fig. 8(b). The DFG is given as an input to any mapping algorithm that can accommodate the delay slot to find a valid mapping. The delay slot is required to schedule the fused nodes with 1 cycle delay after the branch operation. Fig.5(a) shows a valid mapping of the DFG with modulo scheduling. The achieved II=2 which is the lowest among all other techniques.

**Proof of Correctness:** For nodes $x_t, y_t \in N_{if}$ and $x_f, y_f \in N_{else}$, with partial order of $x_t < y_t$ and $x_f < y_f$, meaning $y_t$,$y_f$ cannot be scheduled earlier than $x_t$,$x_f$. An incorrect pairing is $\langle x_t, y_f \rangle$ and $\langle y_t, x_f \rangle$ as shown in fig.6(d). Since the algorithm starts pairing from the terminating nodes $\langle y_t, y_f \rangle$ of either path, and proceeds iteratively through the partial order forming another pair, $\langle x_t, x_f \rangle$, there is no possibility of breaking the partial order and obtaining an incorrect pairing. **Time Complexity** is $O(n)$ where n = $max(|N_{if}|, |N_{else}|) + |N_{other}|$. $O(max(|N_{if}|, |N_{else}|))$ for pairing operations and $O(|N_{other}|)$ for phi elimination.

**Support for Nested Conditionals:** PSB provides maximum performance improvement when the number of operations in the conditional path is large. Hence, for nested conditionals, the formation of fused nodes is done for the outermost conditional block. The no.of operations for the inner nests are typically small and hence are acceptable to be handled by partial predication[10] (preferred over full predication to alleviate the tight restrictions on mapping). The if and else-path operations of the fused nodes are inherently composed of their respective path′s inner conditionals and their operations.

## V. Experimental Results

### A. PSB achieves lower II compared to existing techniques to accelerate control flow

To evaluate the performance of PSB, we have modelled CGRA as an accelerator in Gem5 system simulation framework [18] and integrated our PSB compiler technique as a separate pass in the LLVM compiler framework [19]. The DFG obtained after PSB transformation is mapped using REGIMap mapping algorithm [20] modified to accommodate the delay slot required for correct functioning. Computational loops with control flow are extracted from SPEC2006 [15], biobench [14] benchmarks after -O3 optimization in LLVM. We map the loops on a $4 \times 4$ torus interconnected CGRA with sufficient instruction and data memory. Fig. 9 plots the II achieved by

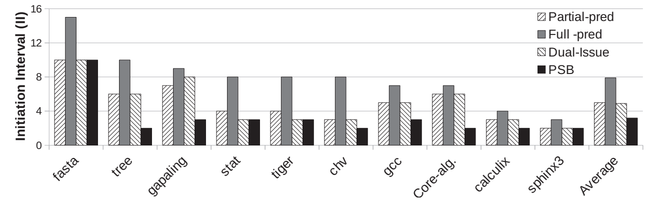


Fig. 9. Performance of compiled loops using i)Partial predication, ii)Full Predication, iii)Dual-Issue, iv) PSB in a 4x4 CGRA. PSB achieves the lowest II. Resource utilization and performance is inversely proportional to II

different techniques. The full predication scheme presented in [10] has the lowest performance due the tight restriction on mapping of operations in the conditional path. Such operations must be mapped only to the PE in which the predicate value is available, which increases the schedule length and ultimately the II. Partial predication scheme performs better since it is devoid of such restrictions and the overhead here is introduction of select operations. Even though the dual issue scheme [12] eliminates execution of unnecessary operations, it suffers from restriction in mapping due to overhead in communicating the predicate to all the merged nodes. The performance improvement of our approach depends on the size of the if-then-else. For the kernels in which the number of operations in the conditional path is more (51% of operations in $tree$, $gapaling$, $gcc$ are in the conditional path) there is a very significant (up to 25% reduction in node size and 45% reduction in edge size on an average due to pairing of operations by PSB) improvement of II - an average of 62% better than other techniques. For benchmarks with smaller if-then-elses, our technique achieves only a moderate reduction in II (only 11% in $sphinx3$, $fasta$, $calculix$). In these cases, the number of operations in the conditional path is relatively low (only 35%) which leads to only moderate reduction in the DFG size (15% and 23% reduction in node and edge size). Therefore, PSB is well suited for loop kernels with relatively large number of operations in the conditional path. By executing operations only from the path taken and eliminating the predicate communication overhead, PSB overcomes the inefficiencies associated with existing techniques, and achieves a performance improvement of 34.6%, 36% and 59.4% on an average compared to the state of the art dual issue scheme [7], partial predication scheme [9] and State based Full Predication (SFP) scheme presented in [10].

### B. PSB architecture has comparable Area and Frequency with existing solutions

We implemented the RTL model of a 4x4 CGRA including the IFU with torus interconnection. Since all PEs have symmetrical interconnections, a single designated PE is connected to the IFU in PSB architecture. A mapping generated for a generic 4x4 CGRA template can be panned across the CGRA template so as to allocate the branch operation to the designated PE. This is not a restriction in mapping since we are able to utilise the symmetry of interconnection. For multiple independent branches, predicates can be communicated to the designated PE through predicate network and then to the IFU. The RTL models were synthesized in 65nm node using RTL

TABLE I
CGRA PLACE AND ROUTE RESULTS

| CGRA | Partial Predication | Full Predication | Dual Issue | PSB |
|---|---|---|---|---|
| Area (sq.um) | 375708 | 384539 | 411248 | 384154 |
| Frequency (MHz) | 463 | 477 | 454 | 458 |

compiler tool, functionally verified and placed and routed using Cadence Encounter. Results are tabulated in table I. PSB architecture does not incur any significant hardware overhead.

### C. PSB has lower energy consumption compared to existing techniques

To evaluate energy consumption, we estimate the dynamic power for each type of PE operation (ALU, routing or IDLE) from [21] and scale to fit our synthesized RTL. Power for an instruction fetch operation for a configuration cache of size 2kb, in 65 nm node, is obtained from cacti 5.3 tool [22]. The total energy spent in executing kernel of each benchmark is modelled as the function of the energy spent per PE per cycle depending upon the type of operation and the instruction fetch power. Fig. 10 shows that the full predication scheme presented in [10] has the highest energy consumption in spite of sleeping the PEs during the execution of the false path. This is due to the higher II caused by tight restrictions in mapping resulting in more instructions fetched and more PEs occupied for execution of the kernel, which leads to a corresponding increase in instruction fetch operation and PE static power. In dual issue scheme, there is an overhead (53% more power) in instruction fetch operation since the number of configuration bits fetched per cycle is twice as much as compared to other techniques. Moreover, this is worsened by the higher II achieved due to predicate communication overhead, increasing the overall number of instruction bits fetched and hence the higher energy consumption per kernel. Even though partial predication scheme executes unnecessary operations, the energy expenditure is compensated to some extent by achieving lower II compared to SFP and dual issue scheme. PSB avoids fetching and executing of unnecessary instructions also achieves the lowest II and hence has the least energy consumption among all techniques. Experimental results show that PSB has 52.1%, 53.1% and 33.53% lower energy consumption on an average compared to state of the art dual issue scheme, full predication and partial predication schemes respectively.

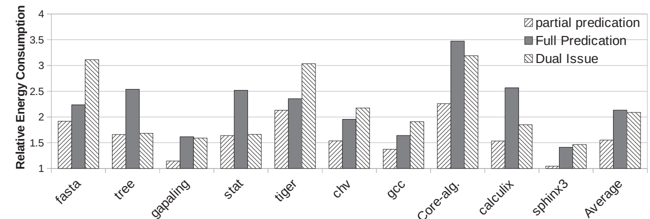



Fig. 10. Relative energy consumption with respect to PSB for executing the the kernel of each benchmark.

## VI. SUMMARY

In this paper, we propose a novel solution to accelerate control flow loops by communicating the branch outcome to the IFU. We eliminate fetching and execution of unnecessary operations and also the overhead due to predicate communication thus overcoming the inefficiencies associated with existing techniques so as to improve the acceleration obtained.

## ACKNOWLEDGMENT

The authors would like to acknowledge the support of National Science Foundation grants CCF-0916652, IIP-1343436, 1055094 (CAREER), IIP-1361926, ASU Center for Embedded Systems, and the Science Foundation Arizona Grant SRG 0211-07.

## REFERENCES

[1] E. Chung, P. Milder *et al.*, "Single-Chip Heterogeneous Computing: Does the Future Include Custom Logic, FPGAs, and GPGPUs?" in *MICRO 2010*, pp. 225–236.

[2] B. Betkaoui *et al.*, "Comparing performance and energy efficiency of FPGAs and GPUs for high productivity computing," in *FPT 2010*, pp. 94–101.

[3] S. Che, J. Li, J. Sheaffer *et al.*, "Accelerating Compute-Intensive Applications with GPUs and FPGAs," in *SASP 2008*, pp. 101–107.

[4] G. Theodoridis *et al.*, "A survey of coarse-grain reconfigurable architectures and cad tools," in *Fine- and Coarse-Grain Reconfigurable Computing*, S. e. Vassiliadis, Ed. Springer, 2007, pp. 89–149.

[5] B. De Sutter *et al.*, *Handbook of Signal Processing Systems*, 2nd ed. Springer, 2013, ch. Coarse-Grained Reconfigurable Array Architectures, pp. 553–592.

[6] A. Carroll *et al.*, "Designing a Coarsegrained Reconfigurable Architecture for Power Efficiency," in *Department of Energy NA-22 University Information Technical Interchange Review Meeting*, 2007.

[7] M. Hamzeh, A. Shrivastava, and S. Vrudhula, "Branch-Aware Loop Mapping on CGRAs," ser. DAC '14. ACM, pp. 107:1–107:6.

[8] S. Mahlke, D. Lin *et al.*, "Effective Compiler Support For Predicated Execution Using The Hyperblock," in *MICRO 25*, 1992, pp. 45–54.

[9] S. Mahlke *et al.*, "A comparison of full and partial predicated execution support for ILP processors," in *Computer Architecture Symposium, 1995.*, pp. 138–149.

[10] K. Han, K. Choi *et al.*, "Compiling control-intensive loops for CGRAs with state-based full predication," in *DATE 2013*, pp. 1579–1582.

[11] K. Chang and K. Choi, "Mapping control intensive kernels onto coarse-grained reconfigurable array architecture," in *ISOCC 2008*, pp. 362–365.

[12] K. Han, J. K. Paek *et al.*, "Acceleration of control flow on CGRA using advanced predicated execution," in *FPT 2010*, pp. 429–432.

[13] K. Han, J. Ahn, and K. Choi, "Power-Efficient Predication Techniques for Acceleration of Control Flow Execution on CGRA," *ACM Trans. Archit. Code Optim.*, vol. 10, no. 2, pp. 8:1–8:25, 2013.

[14] K. Albayraktaroglu, A. Jaleel *et al.*, "BioBench: A Benchmark Suite of Bioinformatics Applications," in *ISPASS 2005*, pp. 2–9.

[15] J. L. Henning *et al.*, "SPEC CPU2006 Benchmark Descriptions," *SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, 2006.

[16] B. R. Rau, "Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops," ser. MICRO 27. ACM, 1994, pp. 63–74.

[17] R. Johnson and K. Pingali, "Dependence-based Program Analysis," *SIGPLAN Not.*, vol. 28, no. 6, pp. 78–89, 1993.

[18] N. Binkert, Beckmann *et al.*, "The Gem5 Simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, 2011.

[19] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *CGO'04*.

[20] M. Hamzeh, A. Shrivastava, and S. Vrudhula, "REGIMap: Register-aware application mapping on Coarse-Grained Reconfigurable Architectures (CGRAs)," in *DAC 2013*, pp. 1–10.

[21] Y. Kim, J. Lee *et al.*, "Improving Performance of Nested Loops on Reconfigurable Array Processors," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, pp. 32:1–32:23, 2012.

[22] H. CACTI, "HP Laboratories Palo Alto, CACTI 5.3," 2008.